**Goal: all the file of rocksdb should be flushed to SSD with the phase 1.0 system call interface. For our requirements, we should implement a customized KVENV especially for the interface in NewSequentialFile, NewWritableFile etc.**

**we should implement a customized KVENV, especially for the interface in NewSequentialFile, NewWritableFile etc.**

**What it means the KVEnv of rocksdb is based on the system call of phase 1.0(put., get etc. interface). In other words, all the data of rocksdb should be flushed to SSD with the phase 1.0 system call interface.**

**Findings:**

* **Env\_hdfs details:**

env\_hdfs is an implementation of the RocksDB Env interface that allows RocksDB to read and write data to and from the Hadoop Distributed File System (HDFS).

The structure of env\_hdfs is designed to provide an interface between RocksDB and HDFS. It includes several key components, such as:

1. A FileSystem object that represents the HDFS file system and provides an interface for opening, reading, writing, and closing files.
2. A WritableFile object that represents a file opened for writing and provides an interface for appending data, syncing data to disk, and closing the file.
3. A SequentialFile object that represents a file opened for reading and provides an interface for reading data from the file.
4. A RandomAccessFile object that represents a file opened for random access and provides an interface for reading and writing data at specific offsets.
5. A Directory object that represents a directory in the HDFS file system and provides an interface for listing files in the directory.

These components are used by RocksDB to read and write data to and from HDFS. When RocksDB needs to read or write data, it calls the appropriate methods on the env\_hdfs objects to perform the necessary operations.

Overall, env\_hdfs provides a convenient and efficient way to use RocksDB with HDFS, allowing for scalable and reliable data storage and retrieval in a distributed computing environment.

* **Now our own kv interface designing steps are:**

Designing an env kv interface in RocksDB to integrate RocksDB with SSD using ioctl direct read write would involve implementing a custom RocksDB environment (Env) that utilizes the ioctl system call for direct read and write operations on the SSD. Here's a high-level overview of the steps involved:

1. Define the interface: First, define the interface for your custom RocksDB Env implementation that allows RocksDB to access the SSD using the ioctl system call. The interface should provide methods for reading, writing, and syncing data to the SSD.
2. Implement the interface: Implement the interface using the ioctl system call to directly read and write data to the SSD. You can use the open(), read(), write(), and close() functions to perform these operations.
3. Optimize for performance: Since SSDs have different performance characteristics than traditional hard disk drives, it's important to optimize your implementation for performance. For example, you may want to use a larger buffer size for writing to the SSD to reduce the number of system calls required.
4. Test and iterate: Once you've implemented the Env KV interface, test it thoroughly to ensure that it works as expected. You may need to iterate on the design and implementation to optimize performance and fix any bugs that you encounter.

**Here are the Details of the steps:**

1. **Defining the interface:**

The KVEnv interface in RocksDB is a key-value store environment that allows RocksDB to write to and read from different types of storage devices. The interface is implemented as a C++ class and provides an abstraction layer over the underlying storage device, allowing RocksDB to use different storage backends without modification.

The KVEnv interface defines a set of methods that RocksDB can use to interact with the storage backend. These methods include:

* NewSequentialFile: Opens a new file for reading.
* NewRandomAccessFile: Opens a new file for reading and seeking.
* NewWritableFile: Opens a new file for writing.
* FileExists: Checks if a file exists.
* GetChildren: Lists the children of a directory.
* DeleteFile: Deletes a file.
* CreateDir: Creates a new directory.
* DeleteDir: Deletes a directory.
* GetFileSize: Returns the size of a file.
* RenameFile: Renames a file.
* LockFile: Locks a file.
* UnlockFile: Unlocks a file.
* GetTestDirectory: Returns a directory that can be used for testing.

RocksDB provides a default implementation of the KVEnv interface that uses the operating system's file system for storage. This implementation is based on the stdio library and is used by default if no other implementation is provided.

However, it's also possible to use custom implementation of the KVEnv interface to use different storage backends. For example, you can use the KVEnv interface to write to and read from an SSD,

**KVEnv interface to write to and read from an SSD details :**

To use the KVEnv interface to write to and read from an SSD, you would need to create a custom implementation of the KVEnv interface that uses the low-level I/O commands provided by the SSD's controller. The main idea is to use the SSD's controller to communicate with the SSD and interact with it as if it were a regular file system.

One way to achieve this is to use the ioctl system call in C to send commands to the SSD's controller. The ioctl system call allows you to perform various operations on a file descriptor, such as reading or setting the device's configuration, or sending special commands to the device.

To write to the SSD using the KVEnv interface and ioctl, you would need to create a new file using the NewWritableFile method and then use the ioctl system call to send a write command to the SSD's controller. The write command would specify the location on the SSD where the data should be written and the data itself.

To read from the SSD using the KVEnv interface and ioctl, you would need to open an existing file using the NewSequentialFile or NewRandomAccessFile method and then use the ioctl system call to send a read command to the SSD's controller. The read command would specify the location on the SSD where the data should be read from and the size of the data to be read.

It's important to note that the implementation will depend on the specific SSD controller you are using and the commands it supports. Also, it's important to keep in mind that different SSD controllers may have different performance characteristics and may require different levels of optimization.

Additionally, it's worth to mention that SSDs have their own wear leveling mechanism which are handled by the SSD firmware and you should be aware of that when implementing this kind of approach.

**RocksDB KVEnv interface to sync database into ssd:**

The RocksDB KVEnv interface provides a way to synchronize the database with an SSD by flushing the data from memory to disk. The Sync method of the KVEnv interface can be used to ensure that all data written to the database is also written to disk.

One way to implement this is to create a custom implementation of the KVEnv interface that uses the low-level I/O commands provided by the SSD's controller to flush data to the SSD.

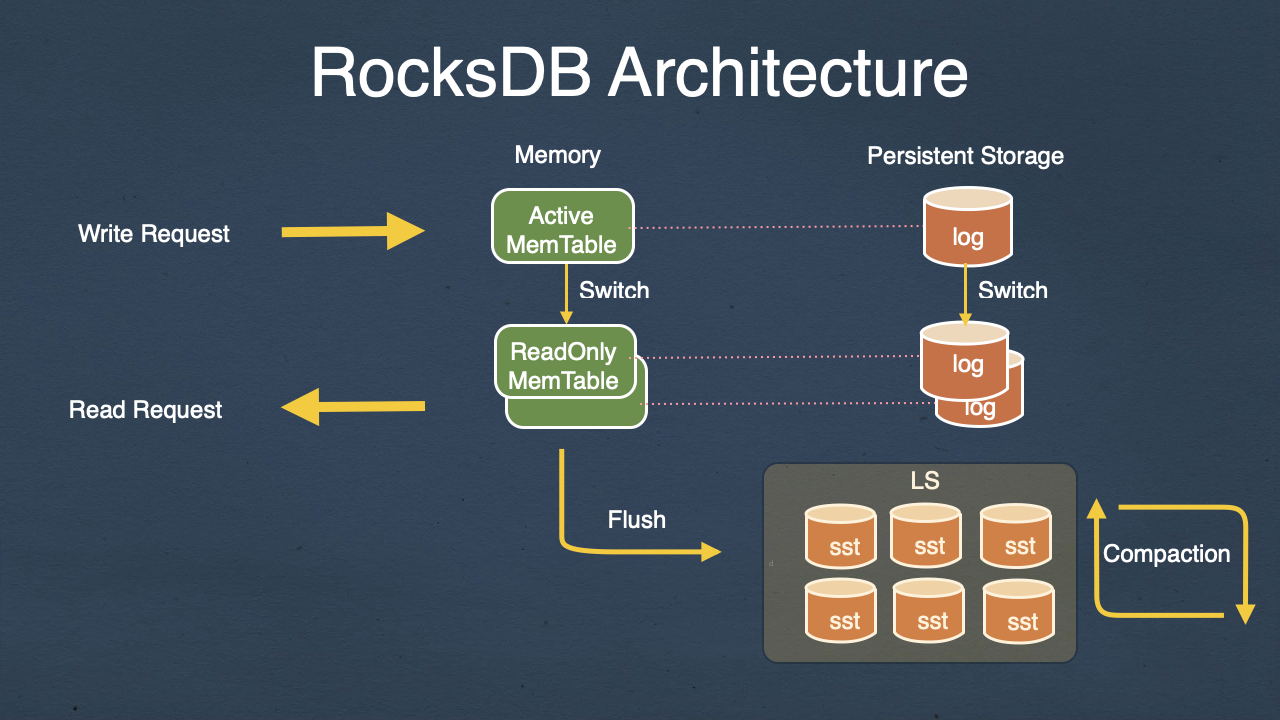
You can use the ioctl system call in C to send commands to the SSD's controller, specifying the location on the SSD where the data should be written and the data itself.

It's important to note that this method may have a performance impact, as it requires the SSD's controller to flush data to the SSD, which can be a time-consuming process. Additionally, it's worth to mention that SSDs have their own wear leveling mechanism which are handled by the SSD firmware and you should be aware of that when implementing this kind of approach.

It's also possible to use the Sync method in conjunction with other RocksDB options such as SyncWAL, which synchronizes the write-ahead-log with the database, to ensure that all data is persisted to disk.

It's important to benchmark and test your implementation to determine the best settings for your specific use case.

1. **Implement the Interface:**

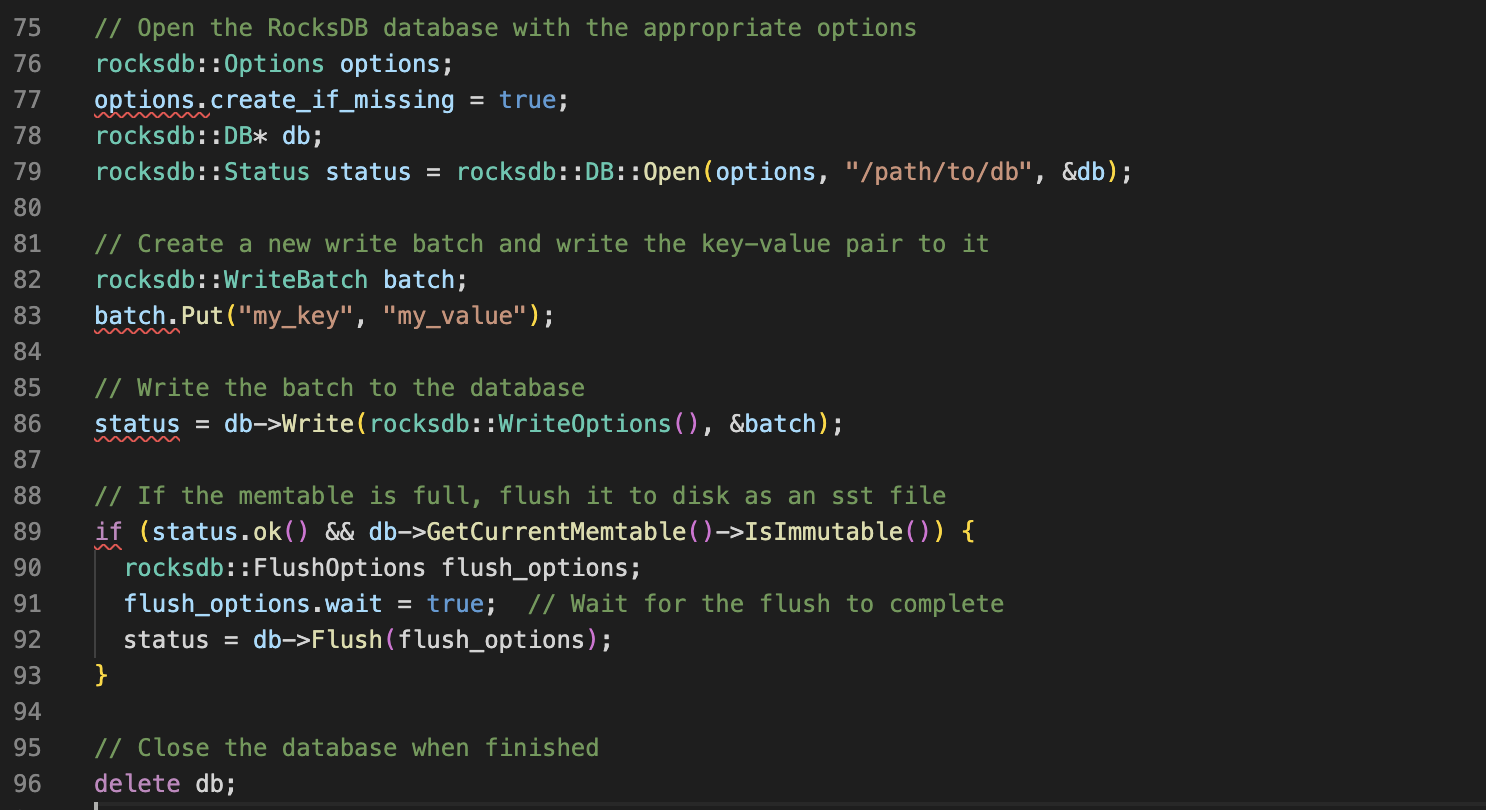
****

When rocksdb\_put is called, the following steps are typically executed:

1. The key-value pair passed to rocksdb\_put is serialized into a format that can be stored in RocksDB's key-value store.
2. RocksDB looks up the key in its internal data structure to determine the location where the new key-value pair should be stored.
3. The serialized key-value pair is written to a log file on disk, which is used to recover the database in case of a crash.
4. The key-value pair is written to a memtable in memory.
5. Once the memtable is full, it is flushed to disk as an sst file.
6. Finally, the function returns a status code indicating whether the operation was successful or not.

Note that the specific implementation details may vary depending on the configuration of the database, such as the choice of compression algorithm and write-ahead log settings.

Step 5 in the rocksdb\_put function involves flushing the memtable to disk as an sst file. Here is some example code that shows how this might be done using the RocksDB C++ API:



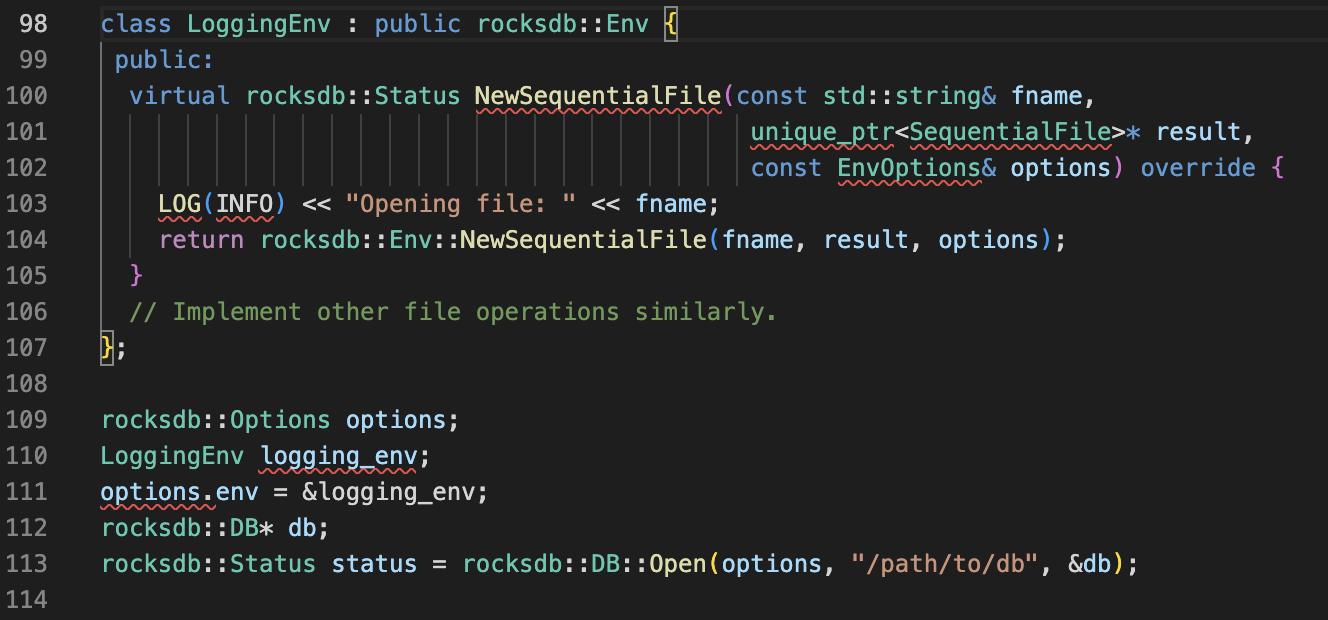
**Automation of env:**

In RocksDB, the env (environment) is responsible for providing an interface between the database and the underlying operating system. It provides an abstraction layer for file and directory operations, such as opening, reading, writing, and closing files.

RocksDB has a default implementation of the env interface that uses the operating system's file system. However, you can also implement your own custom env to provide different functionality, such as using an in-memory file system or an encrypted file system.

When RocksDB is initialized, it automatically creates an instance of the default env implementation, which is used for all file operations by default. However, you can configure RocksDB to use a custom env implementation by passing an instance of your custom env to the Options object when opening the database.

For example, the following code creates a custom env implementation that logs all file operations and then passes it to the Options object:



To use the env\_hdfs.cc interface, you need to link against the HDFS C library and include the hdfs.h header file in your project. You also need to set the rocksdb::Env instance to an instance of HdfsEnv when opening the RocksDB database, as shown in the following example:

rocksdb::Options options;

HdfsEnv hdfs\_env(hdfs\_host, hdfs\_port);

options.env = &hdfs\_env;

rocksdb::DB\* db;

rocksdb::Status status = rocksdb::DB::Open(options, "/path/to/db", &db);

The env\_hdfs.cc file in RocksDB provides an implementation of the rocksdb::Env interface that enables RocksDB to interact with the Hadoop Distributed File System (HDFS). This allows you to use RocksDB as a storage engine for data stored in HDFS.

The env\_hdfs.cc file implements the methods of the rocksdb::Env interface using the HDFS C API. For example, the NewSequentialFile method is implemented as follows:

scss

Copy code

Status HdfsEnv::NewSequentialFile(const std::string& fname, unique\_ptr<SequentialFile>\* result, const EnvOptions& options) { hdfsFile file = hdfsOpenFile(fs\_, fname.c\_str(), O\_RDONLY, 0, 0, 0); if (file == nullptr) { return IOError(fname, errno); } result->reset(new HdfsSequentialFile(file)); return Status::OK(); }

In this implementation, the hdfsOpenFile function is used to open a file in HDFS, and if the file is opened successfully, it is wrapped in a HdfsSequentialFile object and returned via a unique pointer. If the file cannot be opened, an IOError is returned with the appropriate error code.

Similarly, other methods of the rocksdb::Env interface, such as NewRandomAccessFile, NewWritableFile, and FileExists, are implemented using the corresponding HDFS C API functions.

To use the env\_hdfs.cc interface, you need to link against the HDFS C library and include the hdfs.h header file in your project. You also need to set the rocksdb::Env instance to an instance of HdfsEnv when opening the RocksDB database, as shown in the following example:

ruby

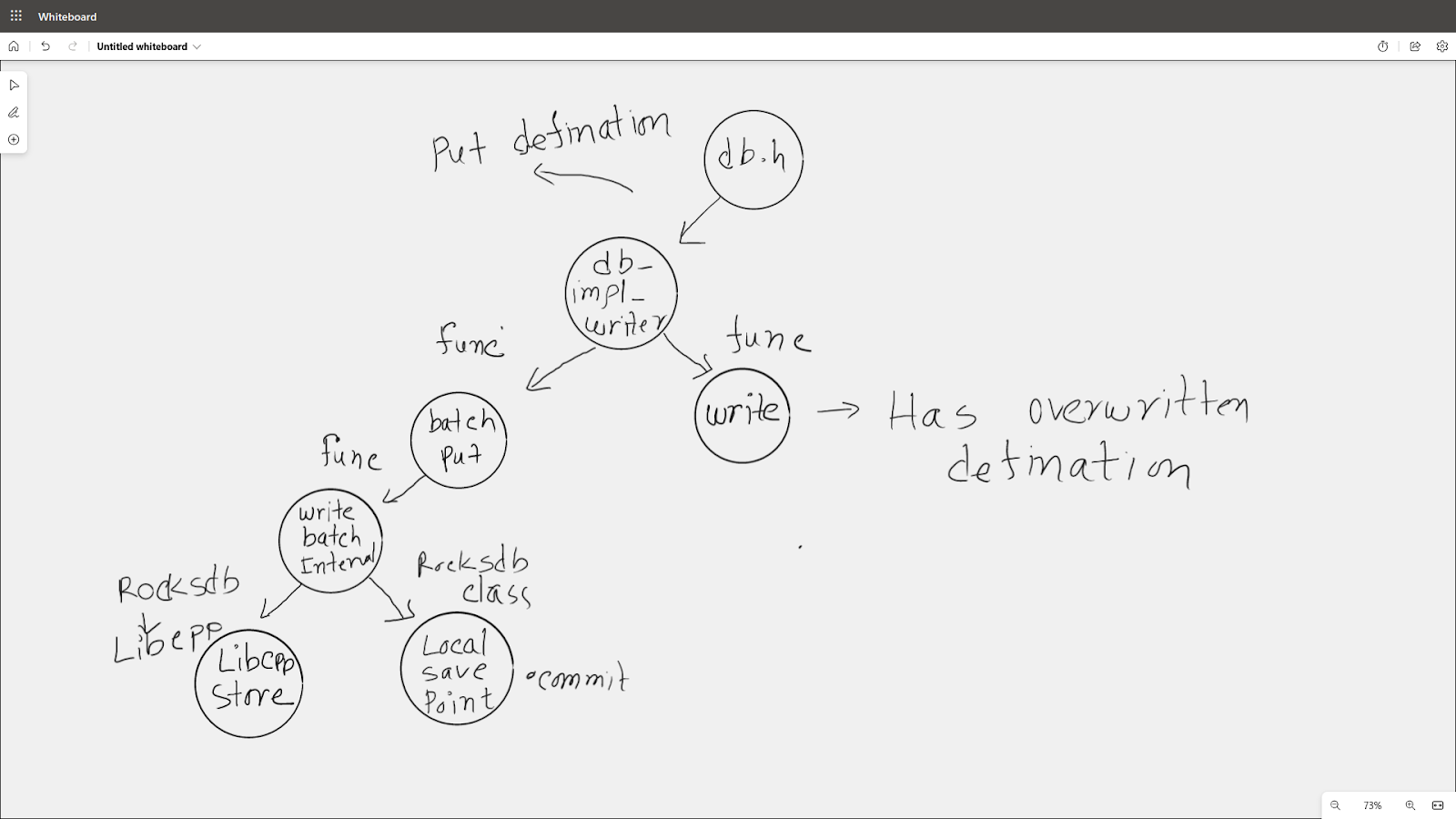
Copy code

rocksdb::Options options; HdfsEnv hdfs\_env(hdfs\_host, hdfs\_port); options.env = &hdfs\_env; rocksdb::DB\* db; rocksdb::Status status = rocksdb::DB::Open(options, "/path/to/db", &db);

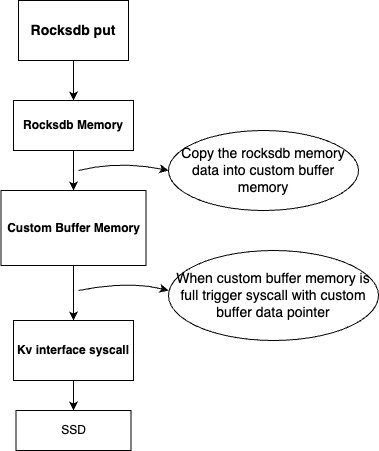
In this example, the HdfsEnv constructor is passed the host and port of the HDFS namenode, and the resulting HdfsEnv object is set as the environment for the RocksDB database using the options.env field.

03/09/2023:

1. **Custom data buffer study:**

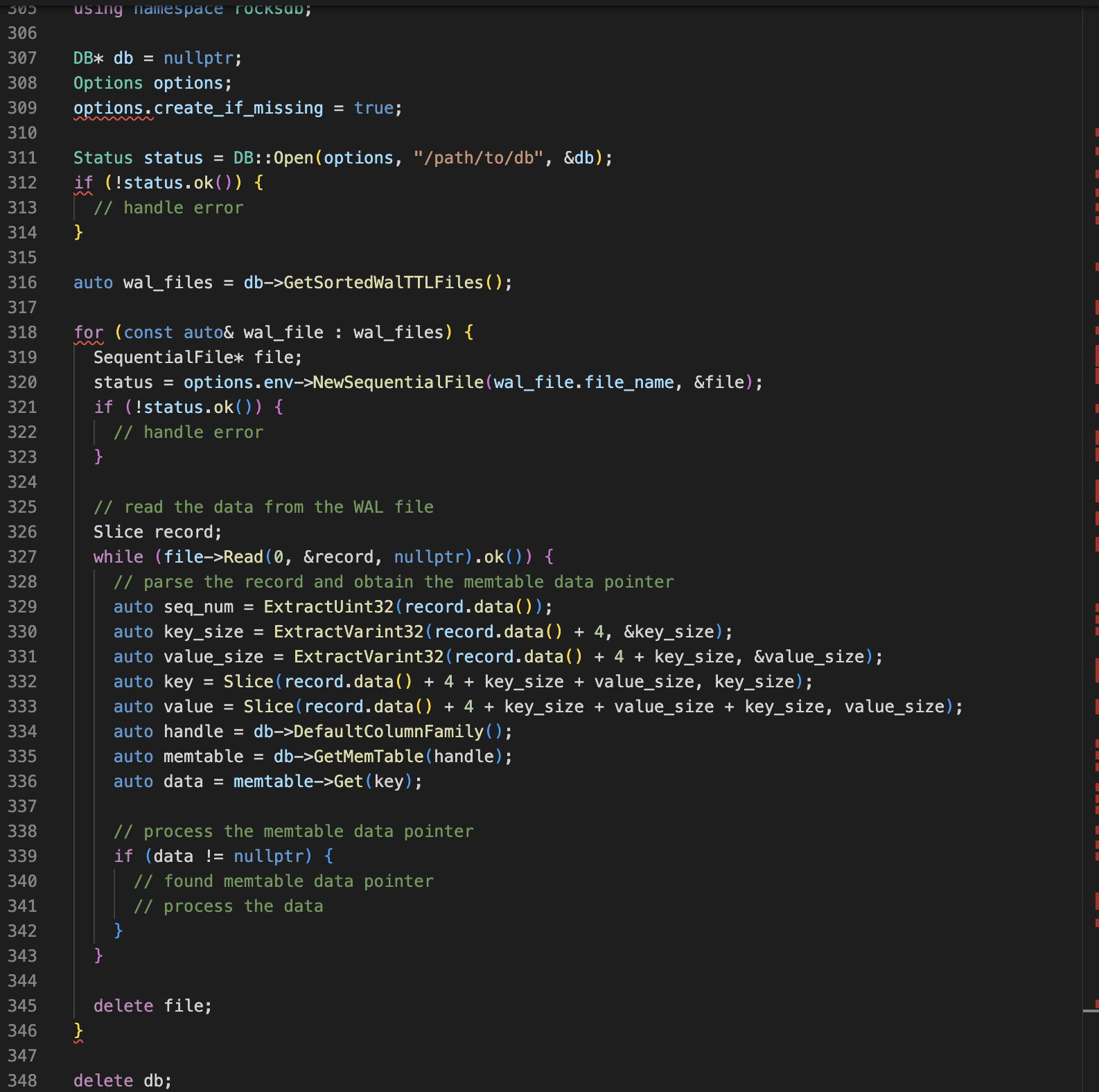


GetSortedWalTTLFiles() method of the RocksDB class. This method returns a list of WalFileInfo objects that contain information about the sorted WAL (Write-Ahead Log) files that contain data that has not yet been flushed to the memtable or to the storage.



Each WalFileInfo object contains a file\_name member that specifies the name of the sorted WAL file, and a min\_seq member that specifies the minimum sequence number of the data contained in the file. You can use this information to locate the data in the sorted WAL file and read it into memory.

Here is an example code snippet that demonstrates how to read the data from the sorted WAL file and obtain the memtable data pointer:



1. **Rocksdb Custom Environment testing:**

Rocksdb custom environment can be provided with the rocksdb::Options property. This is defined as options.env.



This env should override all the virtual functions in rocksdb::Env class. The custom env will inherit all the functions from the base environment class. After inheritance all the virtual functions must be overridden. Otherwise, errors will occur.

My current state of the custom environment:

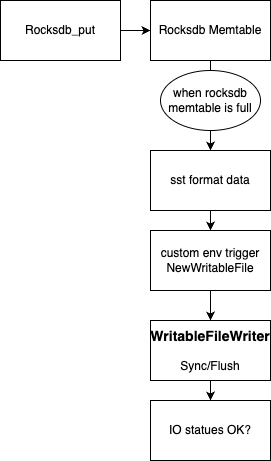
I have inherited the environment class in a custom env class. And overridden the functions with custom functions to see if any of the function triggers when calling put get functions. But when I use this environment I am getting a segmentation fault. Maybe the function implementation is not correct, and that's why this is happening. But when I just chance to default this works.

Segmentation fault: When we override env from rocksdb namespace all it’s pure virtual function(maybe 20+) definition is required but we are using very few that’s why

My next Step:

Env class’s virtual function has been overridden in env\_hdfs.h and composite\_env.h . So I will relate those implementations in our custome environment.

**Summary of the structure:**

****

**osl\_env.cc:**

**#include <fcntl.h>**

**#include <string>**

**#include <sys/mman.h>**

**#include "rocksdb/env\_osl.h"**

**namespace ROCKSDB\_NAMESPACE {**

**Status OSLWritableFile::Append(const Slice& data) {**

**struct csd\_params parameters;**

**parameters.buffer1.command[0]= 'c';**

**parameters.data\_pointer = data.ToString().c\_str();**

**if(!syscall(\_\_NR\_csd\_syscall,(void\*) &parameters)) {**

**printf("Systemcall successfull\n");**

**} else {**

**printf("Error systemcall\n");**

**return Status::IOError("Failed to write to file");**

**}**

**return Status::OK();**

**}**

**Status OSLWritableFile::Close() {**

**if (fd\_ >= 0) {**

**int ret = close(fd\_);**

**fd\_ = -1;**

**if (ret != 0) {**

**return Status::IOError("Failed to close file");**

**}**

**}**

**return Status::OK();**

**}**

**Status OSLWritableFile::Flush() {**

**int ret = fsync(fd\_);**

**if (ret != 0) {**

**return Status::IOError("Failed to flush file");**

**}**

**return Status::OK();**

**}**

**Status OSLWritableFile::Sync() {**

**int ret = fsync(fd\_);**

**if (ret != 0) {**

**return Status::IOError("Failed to sync file");**

**}**

**return Status::OK();**

**}**

**Status OSLEnv::NewWritableFile(const std::string& fname,**

**std::unique\_ptr<WritableFile>\* result,**

**const EnvOptions& options) {**

**if (options.use\_direct\_writes) {**

**printf("Using direct writes\n");**

**}**

**int fd = open(fname.c\_str(), O\_CREAT | O\_WRONLY | O\_TRUNC, 0644);**

**if (fd < 0) {**

**return Status::IOError("Failed to open file for writing");**

**}**

**result->reset(new OSLWritableFile(fd));**

**return Status::OK();**

**}**

**Status OSLEnv::NewSequentialFile(const std::string& fname,**

**std::unique\_ptr<SequentialFile>\* result,**

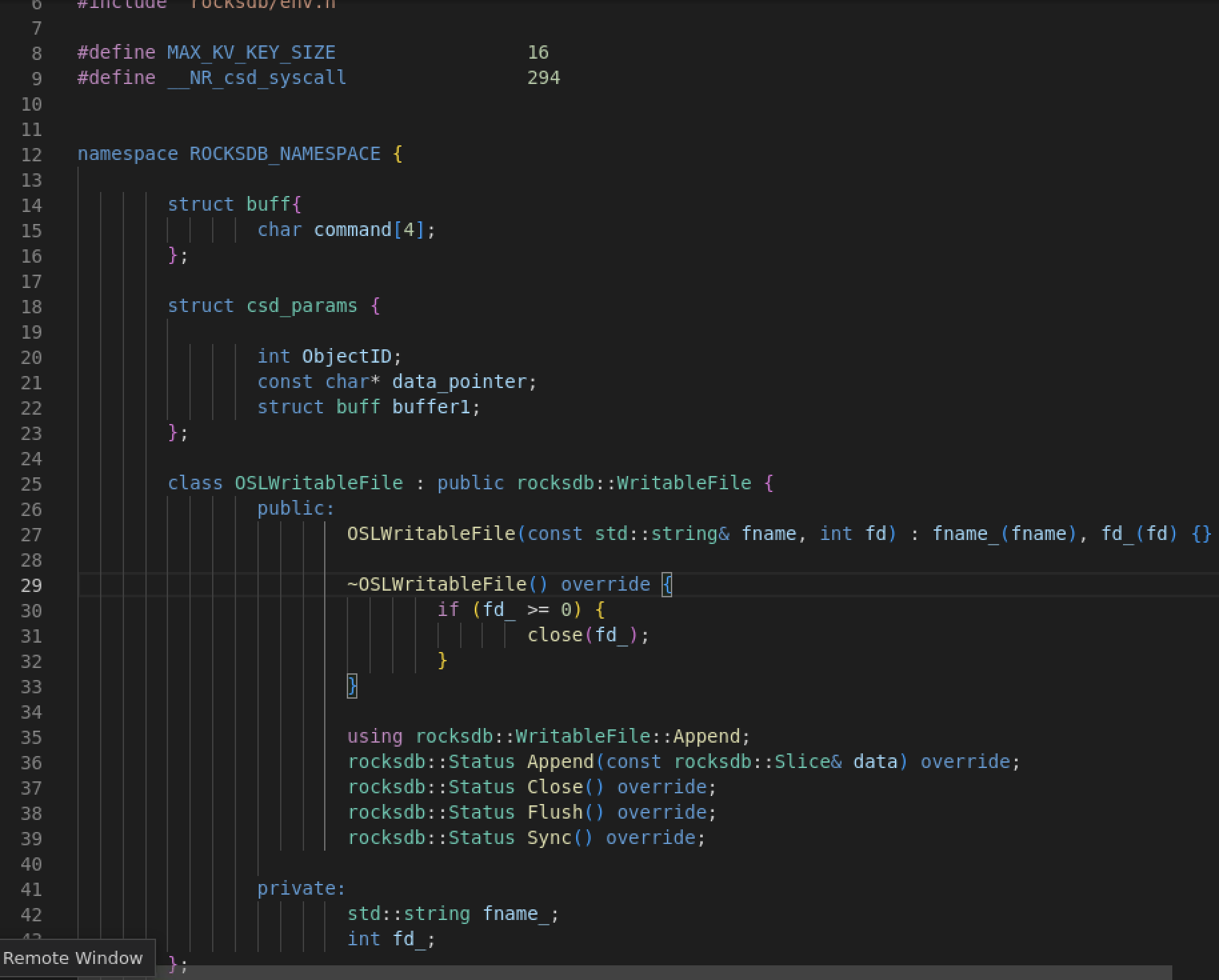
**const EnvOptions& options) {**

**result->reset();**

**}**

**} // namespace ROCKSDB\_NAMESPACE**

osl\_env.h:





Test Program:



Cross-compiling the test program for arm64 architecture and run in the ec2 qemu environment.

**Output:**

